2021년 한국정보보호학회 동계학술대회

CISC-W'21

Conference on Information Security and Cryptography-Winter 2021

2021년 11월 27일 (토)

온라인 컨퍼런스

※개회식, 최우수논문 및 우수 기관장상 논문 발표 촬영 실시간 중계: 가천대학교 글로벌캠퍼스 가천관 B101호

주최 한국정보보호학회 Korea Institute of Information Security & Cryptology

주관 가천대학교 Gachon University

후원 국가정보원 NATIONAL INTELLIGENCE SERVICE, 과학기술정보통신부, 행정안전부, KISA 한국인터넷진흥원, ETRI 한국전자통신연구원 Electronics and Telecommunications Research Institute, NSR 국가보안기술연구소 National Security Research Institute, LG 히다찌, SS 쌍용정보통신주식회사 SsangYong Information & Communications Corp.

한국정보보호학회
Korea Institute of Information Security & Cryptology

게임 엔진 멀티플레이어 프로토콜의 NaN Poisoning 취약점*

하예송† , 최형기‡

성균관대학교 소프트웨어융합대학 (학생, 교수)

NaN Poisoning Vulnerability of Game Engine Multiplayer Protocol*

Ye-Song Ha† , Hyoung-Kee Choi‡

SungKyunKwan University (Undergraduate, Professor)

요 약

게임 엔진의 취약점은 엔진을 사용하여 개발하는 게임들의 취약점으로 이어진다. 본 논문에서는 선행 연구인 'Finding and Exploiting Bugs in Multiplayer Game Engines - DEF CON 28'에서 공개된 Unreal Engine 4의 NaN Poisoning 취약점을 재현하고, 다른 게임 엔진의 멀티플레이어 프로토콜에서 동일한 취약점이 존재하는지 점검한다. Unity HLAPI(High Level API), Unity MLAPI(Mid Level API)를 대상으로 전송 값을 분석한 결과, 현재 NaN Poisoning의 취약점이 없음을 확인하였다. Unity MLAPI는 현재 개발 중인 프로토콜로, 향후 추가되는 기능에 대해 NaN Poisoning 취약점 존재 여부를 검토할 필요가 있다.

I. 서론

게임 엔진은 비디오 게임을 만들기 위한 소프트웨어 개발 환경을 가리킨다. 물리 게임 엔진, 그래픽 엔진, 네트워크 엔진 등 다양한 엔진들이 게임 엔진에 포함되어 있다. 게임 제작 시 다양한 기능을 제공해 편의를 증진시킨다. 게임 엔진이 없는 경우 위 기능들을 직접 프로그래밍해야 하므로, 게임 엔진 사용은 시간과 비용을 절약하게 한다[1].

게임 엔진의 취약점은 엔진을 사용하여 개발하는 게임들의 취약점으로 이어진다. 따라서 게임 엔진의 취약점은 파급력이 크므로, 이를 사전에 방지하는 것이 중요하고, 취약점 발견 시 빠른 조치가 필요하다.

멀티플레이어 게임의 경우 멀티플레이어 프로토콜의 보안도 중요하다. 멀티플레이어 프로토콜은 클라이언트로부터 도착한 게임 액션을 처리하고, 서버는 게임 상태에 대한 주기적인 업데이트를 클라이언트에 전송한다[2]. 멀티플레이어 프로토콜의 보안 사항이 제대로 지켜지지 않을 경우, 세션 탈취, DoS(Denial of Service), 혹은 민감한 정보 노출로도 이어질 수 있다. 선행 연구인 'Finding and Exploiting Bugs in Multiplayer Game Engines - DEF CON 28'에서는 Unreal Engine 4와 Unity에서 멀티플레이어 프로토콜의 버그를 4가지 발견하였다. 임의의 파일 접근, 메모리 노출, 스피드핵, 원격 세션 하이재킹으로, 해당 이슈들은 현재 수정된 상태지만 일부는 구조적 결함으로

* 본 논문은 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임 (No. NRF-2020R1A2C1012708).

† 주저자, ableway98@o365.skku.edu

‡ 교신저자, meosery@skku.edu(Corresponding author)

인한 문제이기 때문에 고쳐지지 않는 버그도 존재한다. 해당 버그들이 다른 게임 엔진에서는 발생하지 않는지, 혹은 발생할 위험성이 있는지 점검의 필요성이 있다.

본 논문에서는 선행 연구의 보안 취약점 중 하나인 'Bug #3 Universal speed hack'을 다룬다. 우선 해당 버그를 재현하는 것을 우선순위로 삼는다. 버그를 재현한 후, 같은 버그가 다른 게임 엔진에서 발생하는지에 대한 여부를 파악한다. 취약점이 존재할 경우 이에 대한 대응책을 연구하고, 예방하는 방법을 고안하는 것을 목표로 한다.

선행 연구는 Unreal Engine 4(이하 UE4), Unity HLAPI(High Level API)를 연구 대상으로 삼았다. 하지만 Unity HLAPI 같은 경우에는 deprecated된 상태이며, 2022년까지 지원 대상이다. Unity HLAPI의 대체재로 Unity MLAPI(Mid Level API)를 오픈소스로 개발 중이고, 실험적 패키지인 v0.1.0이 2021년 3월 24일에 배포되었다[3],[4]. 본 연구에서는 선행 연구 대상인 UE4, Unity HLAPI뿐만 아니라 Unity MLAPI까지 포함한다. 미래에 적용될 멀티플레이어 프로토콜에 대해 보안 연구를 함으로써, 정식으로 Unity MLAPI가 배포되었을 때 보안 향상에 기여할 것으로 기대된다.

II. 선행연구 분석

2.1 NaN Poisoning

본 논문에서는 선행 연구의 취약점 'Bug #3 Universal speed hack'을 다룬다. 해당 취약점은 UE 4.24이하 버전에서 발생한 취약점으로, 부동 소수점 방식의 Not-a-Number(이하 NaN)을 이용한 취약점이다.

부동 소수점 방식은 IEEE 754에서 정의된 실수 표현 방법으로 부호 비트, 지수 비트, 가수 비트로 나누어 표시하는 방법이다. 부동 소수점 방식에는 INF, NaN의 2가지 특수한 값이 있는데, 이 중 NaN은 0÷0이나 음수의 제곱근을 구할 때 발생하는 값이다[4].

NaN 값은 2가지의 특수한 성질을 지닌다. 첫 번째는 NaN 과의 비교 결과는 항상 false가 된다는 것이다. NaN==0, NaN>0, NaN<0, NaN==NaN, 이 모든 비교 결과는 false이다. 두 번째 성질은 NaN propagation이라고도 불리는데, NaN 과의 수학적 연산은 NaN이 된다는 것이다. 예를 들어 NaN+1, NaN-1, NaN×2, NaN÷2, 이 모든 결과값은 NaN이 된다.

2.2 선행 연구 취약점

멀티플레이어 네트워크에서 클라이언트의 캐릭터를 움직이기 위해선 클라이언트에서 서버로 movement RPC(Remote Procedure Call)를 전송해야 한다. 클라이언트에서 movement RPC를 전송 시 서버에서 해당 프로시저가 실행되고, 실행된 결과는 네트워크를 통해서 클라이언트로 전달된다.

UE4에서 movement RPC를 통해 전송하는 값에는 movement vector와 TimeStamp가 있는데, TimeStamp가 float type이므로 NaN을 삽입할 수 있다. 서버에서 수신한 movement RPC를 처리할 때 TimeStamp 유효성을 검사한다.

```
1. float ClientError = DeltaTime - ServerDelta;

2. float NewTimeDiscrepancy =
    ServerData.TimeDiscrepancy + ClientError;

3. ServerData.TimeDiscrepancy = NewTimeDiscrepancy;

4. If(NewTimeDiscrepancy >
    MovementTimeDiscrepancyMaxTimeMargin)
5. {
6.    OnTimeDiscrepancyDetected(NewTimeDiscrepancy);
7. }
```

[그림 1] TimeStamp 유효성 검사 함수

[그림 1]은 TimeStamp의 유효성 검사 함수인 ProcessClientTimeStampForTimediscrepancy의 일부이다. TimeStamp가 NaN일 경우, 해당 함수의 DeltaTime 변수에 NaN 값이 전달된다. 첫 번째 줄에서 DeltaTime이 NaN이므로 ClientError도 NaN이 된다. 두 번째 줄에서 ClientError가 NaN 값을 갖고 있으므로 NewTimeDiscrepancy도 NaN이 되고, 이어서 세

번째 줄의 ServerData.TimeDiscrepancy도 NaN이 된다. ServerData.TimeDiscrepancy는 2번째 줄의 연산에 포함되어 있으므로 한 번 NaN 값이 TimeStamp로 주어지면 NewTimeDiscrepancy는 NaN으로 값이 고정된다. 네 번째 줄에서 NewTimeDiscrepancy이 NaN이므로 비교 결과는 항상 false가 되어 TimeStamp의 변조는 탐지되지 않는다.

해당 취약점은 UE4.25.2에서 수정되었으며, 수신하는 movement RPC의 TimeStamp 값의 NaN 여부를 확인하는 방식으로 수정되었다.

2.3 선행 연구 재현

선행 연구 재현을 위해 UE4.24 버전을 사용하였다. UE4에서 제공하는 'Multiplayer Programming Quick Start Guide'를 참고하여 Demo Game을 제작하였다[5].

```
1. float f=nanf("");

2. ServerMove(
3.     f, NewMove->Acceleration,
4.     …
5. );

6. ServerMove(
7.     0.0001f, NewMove->Acceleration,
8.     …
9. );

10. ServerMove(
11.     NewMove->TimeStamp, NewMove->Acceleration,
12.     …
13. );
```

[그림 2] movement RPC 전송 함수

클라이언트의 CallServerMove 함수에서 movement RPC를 전송하는 코드를 [그림 2]와 같이 변경함으로써 재현했다. 기존의 코드는 serverMove 함수를 1번 호출하여 1개의 movement RPC를 전송한다. 변경한 코드에서는 TimeStamp가 NaN, 0.0001f, 정상 TimeStamp 이렇게 3개의 serverMove를 호출하여 3개의 movement RPC를 한 번에 전송하도록 수정하였다.

[그림 2]로 변경한 클라이언트를 실행 시, 서버에서 TimeStamp의 변조를 탐지하는 변수인 NewTimeDiscrepancy의 값이 NaN으로 고정되어 기능을 잃는다. 이후 0.0001f와 정상 TimeStamp의 RPC를 수신하여 (정상 TimeStamp -0.0001f) 값에 movement vector를 곱한 값만큼 이동이 이루어지게 된다. 플레이어가 기존보다 빠르게 이동하게 되며, 스피드 핵의 효과를 낼 수 있다.

III. 취약점 존재 여부 점검

3.1 Unity HLAPI

본 연구는 Unity 2020.3.11.f1(LTS), Unity HLAPI v1.1.1에서 수행하였다. Unity Technologies에서 배포한 프로젝트 중 'move'를 사용하여 연구를 진행하였다[6]. 클라이언트에서 서버로 전송하는 RPC를 Unity HLAPI에서 Commands라 칭한다. NaN Poisoning 발생 가능성을 점검하기 위해 Commands에서 전송하는 인자를 위주로 분석하였다.

전송 과정에서 호출되는 함수, 관련 dll 파일, 패킷 분석을 수행하였다. 분석 결과 클라이언트에서 서버로 TimeStamp를 전송하지 않고, 전송하는 float type의 변수도 없음을 확인했다. 서버에서 직접 float type의 값을 인자로 전달받도록 함수를 작성하지 않는 경우, Commands를 통해 float type이 전송되지 않으므로 일반적인 환경에서 NaN Poisoning은 불가능하다는 결론을 내렸다. Unity HLAPI의 Commands를 이용해 함수를 작성할 때 float type의 인자를 전달할 경우 NaN Poisoning 발생 가능성이 있으므로 서버에서 해당 인자의 검증을 수행해야 한다.

3.2 Unity MLAPI

본 연구는 Unity 2020.3.11.f1(LTS), Unity MLAPI v0.1.0에서 수행하였다. 연구 수행을 위한 데모 게임으로 Unity MLAPI 공식 문서의 'Hello World'를 사용했다[7]. MLAPI의 RPC는 클라이언트에서 호출, 서버에서 실행하

는 Server RPC와 서버에서 호출, 클라이언트에서 실행하는 Client RPC로 나뉜다. 본 연구에서는 Server RPC에서 전송하는 값 중 TimeStamp가 존재하는지, float type의 값이 있는지 분석하였다.

```
1. namespace MLAPI.Messaging
2. {
3.     internal struct RpcFrameQueueItem
4.     {
5.         …
6.         public float Timestamp;
7.         …
8.     }
9. }
```

[그림 3] TimeStamp 포함 구조체

분석 결과 구조체 RpcFrameQueueItem에서 float type의 TimeStamp를 발견하였다. 해당 TimeStamp는 클라이언트에서 서버로 전송되지 않고, 다른 float type의 변수도 전송하지 않음을 확인하였다. RPC 호출 시 호출 발생 시각을 TimeStamp에 담아 저장하지만, TimeStamp는 저장 외의 다른 용도로 사용되지 않았다. Server RPC를 통해 전송되는 값 중 float type의 값은 존재하지 않으므로 현재 버전에서 MLAPI의 NaN Poisoning의 가능성은 없다.

TimeStamp의 용도에 대해 MLAPI Community에 질문을 올렸으나, MLAPI는 현재 개발 중이고 TimeStamp는 이후 개발될 기능을 위해 추가된 변수라는 답변을 받았다. TimeStamp는 이후 다른 기능을 할 수 있으므로, MLAPI를 꾸준히 tracking 하여 NaN Poisoning 취약점 발생을 방지할 필요가 있다.

IV. 결론

본 논문에서는 선행 연구의 취약점을 UE4에서 재현하고, 해당 취약점이 Unity HLAPI, Unity MLAPI에 존재하는지 점검하였다. TimeStamp를 전송하는 UE4와 다르게, HLAPI와 MLAPI에서는 RPC에 TimeStamp를 전송하지 않았고 따라서 TimeStamp를 통한 NaN Poisoning이 불가능하다는 사실을 확인했다. MLAPI는 현재 개발 중이므로 기능이 추가될 때 이를 tracking 하여 NaN Poisoning의 가능성을 충분히 검토하고 배제해야 한다. RPC 관련 함수를 정의할 때 전달하는 인자로 float type을 사용하면 NaN Poisoning의 가능성이 존재하므로 입력값을 면밀히 검증해야 한다.

[참고문헌]

[1] Dong-Hoon Lee and Jae-Hwan Bae, “Analysis of game engine architecture framework”, Korean Society For Computer Game, Vol. 28, No. 3, p. 51-57, 30 Sep 2015.

[2] Kenesi, Z., et al. “Optimizing Multiplayer Gaming Protocols for Heterogeneous Network Environment.” IEEE International Conference on Communications, Aug 2007.

[2] Don Glover, “UNet Deprecation FAQ”, https://support.unity.com/hc/en-us/articles/360001252086-UNet-Deprecation-FAQ, Apr 2019.

[3] Brandi House, “Evolving multiplayer games beyond UNet”, https://blog.unity.com/technology/evolving-multiplayer-games-beyond-unet, Aug 2018.

[4] V. Rajaraman, “IEEE standard for floating point numbers”. Resonance, vol. 21, pp. 11 - 30 , Feb 2016.

[5] Unreal Engine, “Multiplayer Programming Quick Start Guide”, https://docs.unrealengine.com/4.27/ko/InteractiveExperiences/Networking/QuickStart/, Oct 2021.

[6] Unity Technologies, “UNet Sample Projects”, https://forum.unity.com/threads/unet-sample-projects.331978/, Sep 2014.

[7] Brian Coughlin, “Your First Networked Game "Hello World", https://docs-multiplayer.unity3d.com/docs/tutorials/helloworld/helloworldintro, Sep 2021.